# 1.Web项目

## 1.1使用的缓存是什么？

Caffine cache，是一个本地缓存，Caffine是使用Java8对Guava缓存的重写版本，在Spring Boot2.0中将取代，基于LRU算法实现。官方性能比较在全读全写和75%读和25%写的情况下，Caffine的吞吐率都要比其他缓存，比如Guava，concurrentLinkedHashMap，等要高。

使用方法：LoadingCache<Integer,SchoolInfo> schoolTableCache = Caffine.newBuilder().maximumSize(10000).refreshAfterWrite(1,TimeUnit.HOURS).build(new CacheLoader<Integer,SchoolInfo>()){})

参数含义：maxmumSize(10000):最大长度为10000，表示该缓存中最多容纳的缓存的数量

refreshAfterWrite(1,hours)：表示在一个小时之内如果没有更新或者覆盖，则会在下一次获取该值的时候，会在后台异步地去刷新缓存。如果新的缓存还没有load到，会先返回旧值。

expireAfterWrite：表示某一个缓存在多久之内如果没有更新或者覆盖，就会判断这个缓存失效，先移除key，再重新Load。此时可能会导致没有返回值。

LRU(least recently used)：最近最少使用：使用一个链表。新加进来的缓存被放在链表的表头，刚刚访问过的缓存也被放在链表表头，当链表满的时候，将链表尾部的数据丢弃

## 1.2LRU和LFU的实现

LRU实现:可以使用LinkedHashMap来实现，只需要重写LinkedHashMap中的removeEldestEntry(Map.Entry<> eldest)，在LinkedHashMap中该方法总是会返回false，因此afterNodeInsertion(boolean evict)方法一般不会被执行，但是如果重写removeEldestEntry(Map.Entry<> eldest)，如返回size()>limit；当map的大小大于限制时，就删除双向链表的第一个节点。因为源码中首先有一句first=head 就将自定义的first指向了head。

LinkedHashMap：是在HashMap加入了双向链表的这样一个数据结构，其自定义的Entry节点中除了HashMap都有的key,value,hash,next之外（在构造函数中），还有Entry类型的before和after，用来表示该Entry节点的前向和后向节点，当put,get数据时，除了会像普通的HashMap一样构造数组+链表的结构，还会维护所有节点组成的双向链表结构，LinkedHashMap中还存在head和tail来表示双向链表的头节点和尾节点，put和get数据时，都会把该数据移动到双向链表的尾端，当缓存的容量超过限制值时，通过删除双向链表中head节点的数据来实现。

LFU：通过两层链表以及hashmap来实现，内层链表节点Node存储数据的key,value,frequency，所在的外层节点nq，下一个节点next，前一个节点pre。外层链表节点NodeQueue存储表示下一个节点的next和前一个节点pre，还存储放在外层节点里的Node链表的头节点head和尾节点tail。外层节点按照frequency排序，尾节点为frequency最小的所有节点值。当需要插入数据时，此时该数据的frequency为0，首先根据外层节点的tail来判断其frequency是否为0，如果为0，直接将该数据插入到外层节点的内层链表的头节点处，如果不为0，则新创建一个外层节点，将该外层节点的tail置为该数据node。最后把key和node放入hashmap中。当cache已满需要删除时，直接删除外层链表的tail节点的内层链表的tail节点，此时该节点是频率使用最低，而且访问最少的元素。当访问某个元素的时候，将该元素的frequency加1，并将该元素右移到该节点所在的外层节点的头部节点位置。还需要判断frequency+1后是否与下一个外层节点的frequency相同，如果不相同，需要创建外层节点，并将该元素放入该外层节点的tail处。

<https://www.jianshu.com/p/437f53341f67?tdsourcetag=s_pctim_aiomsg>

缺点：LRU：可能由于一次写入大量冷数据而淘汰大量热点数据

LFU：最近新加入的数据总是先被淘汰掉

一般会联合使用LRFU

## 1.3 Caffine cache的实现原理

<http://www.chinacion.cn/article/5629.html>

<https://www.jianshu.com/p/3434991ad075>

Caffine cache使用的是W-tinyLFU这样一种淘汰机制。在其中使用Count-Min sketch记录访问频率。如果需要记录一个值，首先使用四种hash函数去对其进行hash，再在相应的记录中+1。这样是为了避免冲突。这四个hash函数中选取通过该hash函数得到的位置上的记录中其访问频率最低的那一个hash函数。在caffine cache中会生成一个long型的数组，其大小和缓存的大小接近，但是是2的幂的数。这个数组会记录我们的访问频率。在caffinecache中频率的最大值是15，需要四位来存储，一个long型可以存储16个算法，但是caffine只存储了四种，每个long型被分为四段，每个段里保存的是四个算法的频率。

读写性能为什么比guava cache要好，在guava cache的写过程中还可能会夹杂着国企时间的处理，可能会有淘汰操作，但是caffine里，这些事件是异步操作的。

## 1.4 Caffine cache淘汰机制

数据存储在concurrentHashMap中。caffinecache中有三个记录引用的LRU队列。

Eden队列：在caffine中规定只能为缓存容量的1%。其中记录的是新到的数据防止，防止突发流量由于之前没有访问频率而被淘汰。

Probation队列：缓刑队列，这个队列代表你的数据相对比较冷，马上就要被淘汰了。这个有效大小为size减去eden减去protected。

protected队列，暂时不会被淘汰，但是如果probation队列没有数据了或者protected数据满了，也将会面临被淘汰。需要把probation访问一次后，就会提升为protected。

三个队列的关系：

1.所有的新数据都会进入Eden,

2.Eden如果满了，淘汰进入probation

3.如果在probation中访问了其中某个数据，则这个数据升级为protected

4.如果protected满了就会降级为probation

需要淘汰数据的时候会比较probation队列中的队头和队尾的数据，此时通过Count-Min sketch中记录的频率数据来进行判断：如果队尾大于队头，则队头直接淘汰，如果队尾小于等于5，那么直接淘汰队尾。

## 1.5 使用AOP去拦截异常

[可以使用注解@AfterThrowing](mailto:1.可以使用注解@AfterThrowing)来处理程序中的异常。@AfterThrowing(throwing = “ex”,pointcut = “”)参数ex用来访问目标方法抛出的异常

public …

通知方法

可以使用环绕增强@Around在通知方法里执行目标方法，并catch目标方法抛出的异常。

使用@ControllerAdvice注解来写一个全局的异常处理类。在@ExceptionHander中标注可以捕获的异常的类型，处理异常的方法的参数e表示没有捕获异常的方法中抛出的异常。因此需要自定义异常类，以便通过异常的code和message来获取错误信息返回给前端。在service层定义时应该要抛出对应的异常。

## 1.6 拦截器是怎么实现的

首先需要实现HandlerIntercepter接口，再设置不拦截的路径，设定拦截规则。拦截用户的请求并做相应的处理，可以通过拦截器来进行权限校验，或者是判断用户是否登录等。

jwt使用HS256加密算法。一个JWT分成三部分：头部、荷载、签名。使用.连接。头部是描述JWT的基本信息，有签名所使用的算法。我们使用的是HS256。荷载就是自定义的数据，里面是用户Id，头部和荷载各自base64加密之后使用。签名前两段加入一个密钥用HS256算法形成的。

验证流程：用户登录，服务端验证用户名密码，如果正确，则使用secret生成一个jwt令牌，发送给客户端，客户端再次访问的时候会带上这个jwt令牌，此时服务端使用secret对这个jwt令牌进行校验，如果合法就让用户访问服务器，否则拒绝。

这个secret使用Keys.secretKeyFor()生成

为什么使用jwt不使用token：

服务端验证客户端发送过来的token是需要进行数据查询操作的，但是jwt不用，只用检验。

1. session方式，创建session，存储在数据库中，将具有sessionId的cookie发那个在用户浏览器中，验证sessionid。服务器要为每个用户保留session的信息，连接过多用户会造成服务器内存压力过大。

## 1.7 为什么使用longIntHashMap，而不使用普通的HashMap，性能好在哪里，有没有做过试验？

## 1.8 为什么使用filechannel，而不使用fileStream

## 1.9 ByteBuffer的底层实现，有哪些接口